Chapitre 2

Premiers pas en OpenGL

2.1 Introduction

Ce chapitre doit beaucoup au chapitre 2 de la spécification OpenGL ([SA03]). Le lecteur est invité à s'y reporter pour une description plus précise de la bibliothèque OpenGL.

OpenGL (« Open Graphics Library ») est une interface logicielle pour les matériels graphiques. Cette interface consiste en des procédures et fonctions permettant au développeur de décrire les objets et les opérations intervenant lors du rendu d'images et tout spécialement de scènes 3D. Grâce à cette bibliothèque, la gestion des différentes optimisations présentes dans les cartes graphiques est totalement transparente (ou presque) pour le programmeur. Ainsi, il n'y a pas à se préoccuper du type et de la marque de la carte graphique sur laquelle sera exécuté le programme.

OpenGL est uniquement dédié au rendu (et à la lecture) dans un tampon d'affichage (si vous avez une meilleure traduction pour *frame buffer*, je suis preneur ...). Ainsi le support de la souris ou du clavier, par exemple, devront être gérés par d'autres bibliothèques (comme la GLUT ou encore GTK+).

Dans ce qui suit, une primitive graphique pourra être un point (représenté par un ou plusieurs pixels) ou bien un segment, un polygone ou un rectangle de pixels. La bibliothèque graphique dessine les primitives selon un certain nombre de modes. Ces modes peuvent être modifiés indépendemment les uns des autres. Les réglages de l'un n'affectant pas ceux des autres. Le paramétrage des modes, la définition des primitives et, de manière générale, toute opération graphique sont réalisés par l'envoi de commandes sous la forme de procédures ou de fonctions.

Les primitives sont définies par un ou plusieurs sommets. Des données sont affectées à chaque sommet (comme la position, la couleur, la normale ou encore les coordonnées de texture) et les sommets sont traités dans l'ordre et de la même manière. La seule exception est le cas du découpage où il peut alors y avoir modification des données associées aux sommets et même création de nouveaux sommets.

Les commandes sont toujours exécutées dans l'ordre où elles sont envoyées, cependant le temps entre l'instant où elles sont envoyées et l'instant où elles sont exécutées est indéterminé.

Les données associées aux commandes sont traitées dès la réception de celles-ci. Ceci signifie que même si une commande nécessite un pointeur sur une donnée, la donnée sera récupérée lors de l'appel ; tout changement ultérieur de celle-ci n'aura pas d'effet sur la commande.

Le modèle d'interprétation des commandes OpenGL est le modèle client-serveur. Ceci signifie qu'un programme (le client) émet des commandes qui sont traitées par la machine OpenGL

(le serveur). Le serveur peut être sur une machine distante (mais ce n'est pas obligatoire). Nous désignerons par le terme générique « état de la machine OpenGL » l'ensemble des données nécessaires pour le rendu d'une scène à un instant donné. Un serveur peut contenir plusieurs contextes OpenGL, chacun d'entres eux « encapsule » un état de la machine OpenGL. Un client peut choisir de se connecter à n'importe lequel de ces contextes.

Les effets des commandes OpenGL sur le tampon d'affichage sont controlés à terme par le gestionnaire de fenêtres qui a alloué les ressources pour le tampon d'affichage. C'est le gestionnaire de fenêtres qui détermine quelles sont les portions du tampon d'affichage qui sont accessibles à la machine OpenGL et qui communique à celle-ci la structure du tampon. Ainsi, il n'y a pas de commandes pour configurer le tampon d'affichage. De même l'initialisation d'un contexte OpenGL est réalisée lorsque le gestionnaire de fenêtres alloue une fenêtre pour un rendu OpenGL.

2.2 Syntaxe des commandes OpenGL

Les noms de commandes, constantes et types OpenGL sont tous préfixés (par, respectivement, `gl`, `GL_` et `GL` en C).

Les commandes sont des fonctions ou des procédures. Différentes commandes peuvent effectuer la même opération mais diffèrent par la façon dont leurs sont transmis leurs arguments.

Les commandes sont formées (outre le préfixe GL) d'un nom suivi d'au plus quatre caractères. Le premier caractère, si c'est un chiffre, indique le nombre de valeurs qui doivent être présentes dans la commande. Le caractère ou la paire de caractères suivant indique le type spécifique des arguments : entier (sur 8, 16 ou 32 bits), réel en simple précision ou réel en double précision. Le caractère final, s'il est présent, indique que la commande prend en argument un pointeur sur un tableau de valeurs du type spécifié au lieu d'une liste de valeurs.

La commande **Vertex** existe ainsi en plusieurs «versions», dont :

```
void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

et

```
void glVertex2sv(GLshort v[2]);
```

Plus généralement, la déclaration d'une commande sera de la forme :

```
rtype glNom{ε 1234}{ε b s i f d ub us ui}{ε v}
                ([args, ]T argsl, ..., T argsN[, args]);
```

Les spécifications de types sont résumées au tableau 2.1 tandis que les types définis dans OpenGL sont indiqués dans le tableau 2.2.

2.3 Gestion des erreurs

La machine OpenGL ne détecte que quelques unes des situations qui peuvent être considérées comme des erreurs. Ceci afin de ne pas trop affecter les performances d'un programme sans erreurs.

La commande

```
GLenum glGetError(void);
```

Lettre	Type OpenGL correspondant
b	`GLbyte`
s	`GLshort`
i	`GLint`
f	`GLfloat`
d	`GLdouble`
ub	`GLubyte`
us	`GLushort`
ui	`GLuint`

TAB. 2.1 – *Types des arguments des commandes OpenGL*

Type OpenGL	Nombre minimum de bits	Description
`GLboolean`	1	booléen
`GLbyte`	8	entier signé représenté en complément à 2
`GLubyte`	8	entier non signé
`GLshort`	16	entier signé représenté en complément à 2
`GLushort`	16	entier non signé
`GLint`	32	entier signé représenté en complément à 2
`GLuint`	32	entier non signé
`GLsizei`	32	taille entière non négative
`GLenum`	32	valeur enumérée entière
`GLintptr`	*ptrbits*	entier signé représenté en complément à 2
`GLsizeiptr`	*ptrbits*	taille entière non négative
`GLbitfield`	32	champ de bit
`GLfloat`	32	valeur réelle
`GLclampf`	32	valeur réelle comprise dans $[0, 1]$
`GLdouble`	64	valeur réelle
`GLclampd`	64	valeur réelle comprise dans $[0, 1]$

TAB. 2.2 – *Types OpenGL*

est utilisée pour obtenir des informations sur les erreurs. À chaque erreur détectable est assigné un code numérique. Quand une erreur est détectée, un drapeau est levé et le code enregistré. Si d'autres erreurs apparaissent, elles ne modifieront pas le code enregistré. Lorsque la commande **GetError** est invoquée, le code est retourné et le drapeau abaissé.

Si un appel à **GetError** retourne `NO_ERROR` alors c'est qu'aucune erreur détectable n'est apparue depuis le dernier appel à **GetError** (ou depuis l'initialisation de la machine).

Afin de permettre l'implantation de la librairie sur les machines distribuées, une suite de paires (drapeau,code) peut être générée. Dans ce cas, chaque appel à **GetError** retourne le code d'une paire différente de (drapeau,code) jusqu'à ce que tous les codes `NO_ERROR` soient retournés.

Le tableau 2.3 liste les codes d'erreurs possibles.

Erreur	Description	Effet sur la commande ayant généré l'erreur
GL_INVALID_ENUM	argument enum non valide	la commande a été ignorée
GL_INVALID_VALUE	argument numérique non valide	la commande a été ignorée
GL_INVALID_OPERATION	Opération illégale dans l'état courant	la commande a été ignorée
GL_STACK_OVERFLOW	La commande a du causer un débordement de pile	la commande a été ignorée
GL_STACK_UNDERFLOW	La commande a du dépiler une pile à 1 élément	la commande a été ignorée
GL_OUT_OF_MEMORY	Il ne reste pas assez de mémoire pour exécuter la commande	état de la machine OpenGL indéfini
GL_TABLE_TOO_LARGE	Le tableau spécifié est trop grand	la commande a été ignorée

TAB. 2.3 – *Erreurs OpenGL*

2.4 Le paradigme `Begin/End`

En OpenGL, la plupart des objets géométriques sont définis en incluant une suite de coordonnées de sommets – avec éventuellement d'autres données (couleur, normale, `etc.`) – dans un bloc **Begin/End**. Il y a dix objets géométriques qui peuvent être dessinés de cette façon, dont les points, les lignes polygonales et les polygones.

Chaque sommet est spécifié avec deux, trois ou quatre coordonnées. D'autres données peuvent également être utilisées lors du traitement de chaque sommet, comme la normale courante ou la couleur courante. Les valeurs courantes font partie de l'état de la machine OpenGL.

La façon dont est construite une primitive est schématisée en figure 2.1 (tirée de [SA03], p. 14).

Les deux commandes permettant de définir un bloc **Begin/End** sont :

```
void glBegin(GLenum mode);
void glEnd(void);
```

`mode` peut prendre les dix valeurs suivantes (en fait, la spécification indique qu'il y a une onzième valeur permettant de ne générer aucun objet) :

- `GL_POINTS` : permet de spécifier une suite de points.
- `GL_LINES` : permet de spécifier une suite de segments. Les sommets sont traités par paire, chaque couple de sommets indiquant le début et la fin d'un segment. Si le nombre de sommets est impair alors le dernier sommet est ignoré.
- `GL_LINE_STRIP` : définit une suite de segments reliés entres eux. Le premier sommet indique le début du premier segment. Le i^{e} sommet indique la fin du $i-1^{\text{e}}$ segment et le début i^{e} segment. Le dernier sommet indique la fin du dernier segment. Si un seul sommet est indiqué, aucune primitive n'est générée.
- `GL_LINE_LOOP` : idem que pour `GL_LINE_STRIP` avec l'ajout d'un dernier segment entre le dernier et le premier sommet.
- `GL_POLYGON` : permet de décrire un polygone en donnant son périmètre. Le périmètre est entré sous la forme d'une suite de segments analogue à ce qui est décrit pour `GL_LINE_LOOP`. Aucune primitive n'est générée s'il y a moins de trois sommets. Seuls les polygones convexes

sont correctement gérés. Attention : l'ordre des sommets est important (il permet, entre autre, de distinguer la face avant de la face arrière).

- `GL_TRIANGLES` : permet de spécifier une suite de triangles. Les sommets sont traités par groupe de trois, de même que pour les polygones, l'ordre d'apparition des sommets est important. Les sommets surnuméraires sont ignorés.
- `GL_TRIANGLE_STRIP` : définit une «bande» de triangles. Il s'agit d'une suite de triangles reliés par des côtés adjacents. Les trois premiers sommets indiquent le premier triangle (l'ordre est important). Chaque nouveau sommet définit un nouveau triangle comme indiqué en figure 2.2(b). Si moins de trois sommets sont indiqués, aucune primitive n'est générée.
- `GL_TRIANGLE_FAN` : idem que pour `GL_TRIANGLE_STRIP`, sauf que tous les triangles ont le premier sommet en commun (*cf.* FIG. 2.2(c)).
- `GL_QUADS` : permet de spécifier une suite de quadrilatères. Les sommets sont traités par groupe de quatre. Les sommets surnuméraires soint ignorés.
- `GL_QUAD_STRIP` : définit une «bande» de quadrilatères (*cf.* FIG. 2.3(b)). Si moins de quatre sommets sont indiqués, aucune primitive n'est générée. Si le nombre de sommets est impair alors le dernier sommet est ignoré.

Seul un nombre limité de commandes OpenGL sont autorisées dans un bloc **Begin/End** (nous en verrons quelques unes dans la suite, une liste exhaustive est donnée dans [SA03],p. 19). L'exécution d'une commande non autorisée à l'intérieur d'un bloc **Begin/End** peut générer une erreur du type `GL_INVALID_OPERATION`. Certaines commandes non autorisées ne génèrent pas forcément d'erreur, l'effet sur la machine OpenGL est alors indéfini.

2.5 Spécification des sommets

Les commandes permettant de spécifier les coordonnées des sommets sont :

```
void glVertex{234}{sifd}(T coords);
void glVertex{234}{sifd}v(T coords);
```

Un appel à la commande **Vertex** spécifie les quatre coordonnées homogènes x, y, z et w. Un appel à **Vertex2** fixe les coordonnées x et y, la coordonnée z est implicitement fixée à 0 et w à 1. **Vertex3** affecte les valeurs transmises aux coordonnées x, y et z, la coordonnée w est implicitement fixée à 1.

Les valeurs courantes sont associées à chaque sommet comme données complémentaires. On peut changer ces valeurs courantes à tout moment en utilisant la commande appropriée.

Nous ne détaillerons pas toutes les commandes permettant de changer les valeurs courantes (nous les décrirons au fur et à mesure que les notions associées seront abordées en cours). Nous pouvons néanmoins donner celles permettant de spécifier la normale courante et la couleur courante.

Les commandes

```
void glNormal3{bsifd}(T coords);
void glNormal3{bsifd}v(T coords);
```

permettent de fixer la valeur de la normale courante. Les arguments sont convertis en valeurs réelles comme indiquées dans le tableau 2.4.

Type OpenGL	Conversion
ubyte	$c/(2^8 - 1)$
byte	$(2c + 1)/(2^8 - 1)$
ushort	$c/(2^16 - 1)$
short	$(2c + 1)/(2^16 - 1)$
uint	$c/(2^32 - 1)$
int	$(2c + 1)/(2^32 - 1)$
float	c
double	c

TAB. 2.4 – *Conversion des type OpenGL en représentation réelle interne*

Nous verrons au chapitre 5 comment gérer la couleur des objets lorsqu'on utilise des lumières. Cependant, si le calcul de l'éclairement est désactivé (ce qui est le cas par défaut), on peut fixer la couleur courante grâce à la commande :

```
void glColor{34}{bsifd ubusui}(T comps);
void glColor{34}{bsifd ubusui}v(T comps);
```

La commande **Color3** fixe les composantes R (rouge), G (vert) et B (bleu) de la couleur courante. La composante A (alpha) est implicitement fixée à 1.0. De même, la commande **Color4** fixe les composantes R, G, B et A de la couleur courante.

Depuis la version 1.4 d'OpenGL, nous disposons également d'une couleur secondaire (*grosso-modo* il s'agit de la partie de la couleur de l'objet qui n'est pas affectée par le texturage, nous n'en dirons pas plus dans ce cours). Elle peut être fixée par la commande

```
void glSecondaryColor3{bsifd ubusui}(T comps);
void glSecondaryColor3{bsifd ubusui}v(T comps);
```

Il n'existe qu'une version à trois paramètres de la commande **SecondaryColor**. En fait, le canal alpha de la couleur secondaire n'est jamais utilisé.

Les arguments transmis aux commandes **Color** et **SecondaryColor** sont convertis en valeurs réelles comme indiqué dans le tableau 2.4. Une valeur 0.0 correspond à la valeur minimale qu'une composante du tampon d'affichage peut prendre, un 1.0 correspondant à la valeur maximale.

Bien sûr les commandes ci-dessus ne fonctionnent que si la machine OpenGL est en mode de couleur RGBA. Il existe également un mode de couleurs indexées. Le mode dans lequel est la machine OpenGL est choisi lors de l'initialisation de celle-ci. En mode indexé, la commande

```
void glIndex{sifd ub}(T index);
void glIndex{sifd ub}v(T index);
```

permet de sélectionner l'index de la couleur courante.

2.6 Les matrices en OpenGL

2.6.1 Opérations sur les matrices

Pour rendre une scène 3D, OpenGL utilise deux transformations de matrices respectives `GL_MODELVIEW` et `GL_PROJECTION`. À chaque sommet est appliqué la matrice `GL_MODELVIEW` afin de lui appliquer les transformations le plaçant dans la scène 3D, puis on applique au point obtenu la matrice `GL_PROJECTION` afin de le projeter sur un plan (permettant ainsi la représentation de la scène en 2D). Le lecteur est invité à se reporter au chapitre 3 pour plus de détails concernant ces opérations.

Lorsqu'on souhaite effectuer des opérations sur une matrice, on doit d'abord indiquer à la machine OpenGL sur quelle matrice vont porter ces opérations. En effet de même qu'il y a une couleur courante ou une normale courante, il y a une matrice courante. La commande permettant de sélectionner la matrice courante est :

```
void glMatrixMode(GLenum mode);
```

`mode` peut prendre la valeur `GL_MODELVIEW` ou `GL_PROJECTION`. Il existe une autre matrice permettant de calculer les coordonnées dans les textures : `GL_TEXTURE`. Depuis la version 1.2 d'OpenGL, on trouve également la matrice `GL_COLOR`.

Les deux commandes de base pour travailler sur les matrices sont :

```
void glLoadMatrix{fd}(T m[16]);
void glMultMatrix{fd}(T m[16]);
```

La commande **LoadMatrix** remplace la matrice courante par la matrice transmise en paramètre. Celle-ci est donnée sous la forme d'un tableau de 16 valeurs consécutives représentant les vecteurs colonnes de la matrices :

$$\begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix}$$

La commande **MultMatrix** multiplie à droite la matrice courante par la matrice transmise (de la même façon que pour **LoadMatrix**). Ainsi si C est la matrice courante et T la matrice transmise, la matrice courante C' après l'appel à la commande **MultMatrix** sera :

$$C' = C.T.$$

Les commandes

```
void glLoadTransposeMatrix{fd}(T m[16]);
void glMultTransposeMatrix{fd}(T m[16]);
```

ont le même effet que les commandes **LoadMatrix** et **MultMatrix**. Seul change la façon dont est transmise la matrice paramètre, cette fois-ci le tableau est la suite des vecteurs lignes :

$$\begin{pmatrix} m[0] & m[1] & m[2] & m[3] \\ m[4] & m[5] & m[6] & m[7] \\ m[8] & m[9] & m[10] & m[11] \\ m[12] & m[13] & m[14] & m[15] \end{pmatrix}$$

La commande

```
void glLoadIdentity(void);
```

remplace la matrice courante par la matrice identité :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Il existe 5 autres commandes permettant d'agir sur les matrices. Il s'agit de transformations affines. En fait, OpenGL calcule la matrice correspondant à la transformation puis utilise la commande **MultMatrix**.

La commande

```
void glRotate{fd}(T θ, T x, T y, T z);
```

correspond à une rotation d'angle θ (dans le sens trigonométrique) autour de la ligne passant par O et de vecteur directeur (x, y, z).

La commande

```
void glTranslate{fd}(T x, T y, T z);
```

correspond à une translation de vecteur (x, y, z).

La commande

```
void glScale{fd}(T x, T y, T z);
```

correspond à une mise à l'échelle de paramètres x, y et z.

Deux autres transformations correspondant à des projections (**Frustum** et **Ortho**) seront décrites au chapitre 3.

2.6.2 Les piles de matrices

À chaque matrice courante est associée une pile. Pour la matrice `GL_MODELVIEW` cette pile peut contenir au moins 32 matrices. Pour les autres matrices la hauteur de la pile peut au moins atteindre 2. La matrice courante est en fait la matrice qui se trouve au sommet de la pile courante. Pour empiler la matrice courante sur la pile courante on fait appel à la commande :

```
void PushMatrix(void);
```

Pour dépiler la matrice de la pile on utilise :

```
void PopMatrix(void);
```

Dépiler une pile à un élément génère une erreur du type `GL_STACK_UNDERFLOW` et empiler sur une pile pleine provoque l'erreur `GL_STACK_OVERFLOW`.

2.7 Le tampon d'affichage

Nous terminerons cette introduction à OpenGL par trois commandes permettant de gérer le tampon d'affichage.

La commande

```
void glClear(GLbitfield buf);
```

permet d'affecter la même valeur à tous les pixels d'un ou plusieurs tampons. La valeur `buf` est le résultat d'un OU bit à bit de constantes indiquant quels tampons doivent-être remplis. Les constantes `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_STENCIL_BUFFER_BIT` et `GL_ACCUM_BUFFER_BIT` désignent respectivement les tampons d'affichage, de profondeur, pochoir et d'accumulation.

La commande

```
void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a);
```

permet de fixer la valeur d'effaçage pour le tampon d'affichage losqu'on est en mode RGBA. En mode indexé, il faut utiliser la commande :

```
void glClearIndex(GLfloat index);
```

Il existe, bien sûr, des commandes analogues pour les tampons de profondeur, pochoir et d'accumulation (respectivement **ClearDepth**, **ClearStencil** et **ClearAccum**).

Enfin, la commande

```
void glFlush(void);
```

permet de forcer l'exécution des commandes OpenGL en attente.

2.8 Introduction à la GLUT

Nous avons vu qu'OpenGL se contente de fournir des fonctions de rendu. Il faut donc utiliser une API différente pour la gestion de l'interface fenêtrée (ouverture des fenêtres, gestion du clavier et de la souris, *etc.*). De nombreux gestionnaires de fenêtres permettent de gérer l'API OpenGL (Windows, Gnome, KDE). Cependant, utiliser les API de ces gestionnaires nécessiterait plus qu'une rapide présentation dans ce chapitre. Aussi, pour réaliser nos programmes, nous nous contenterons d'une API minimaliste qui est implémentée sous de nombreux gestionnaires : l'« OpenGL Utility Toolkit » (par la suite, nous l'appelerons par son petit nom : la GLUT). Cette API sera certainement trop limitée pour réaliser des applications avancées mais elle sera amplement suffisante dans le cadre de ce cours. Elle permet en outre de ne pas surcharger l'écriture des exemples.

L'utilisation de la GLUT se décompose en trois étapes :

- Création de la fenêtre
- Initialisation du contexte OpenGL
- Spécification des fonctions de traitement des évènements (les anglophones parlent de « callback functions »)

– Boucle d'attente d'évènements

Nous donnons par la suite un bref descritif des fonctions principale de la GLUT. Le lecteur est invité à consulter la spécification de la GLUT [Kil96] pour connaître un descriptif exhautstif des possibilités offertes par cette interface de programmation.

2.8.1 La fenêtre

L'initialisation d'une fenêtre se fait grâce à cinq fonctions.

– `glutInit(int *argc, char **argv)` : initialise les données nécessaires au fonctionnement de la GLUT. Cette fonction traite les options qui lui sont éventuellement transmises par les arguments de la ligne de commande. Cette fonction doit être appelée avant toute autre fonction de la GLUT.

– `glutInitDisplayMode(unsigned int mode)` : permet d'indiquer certaines propriétés du contexte OpenGL qui sera associé à la fenêtre. Ainsi cette fonction permet de décider si on utilisera des couleurs indexées ou un modèle RGBA, si on utilisera un seul tampon d'affichage ou une fenêtre à deux tampons ou encore si on utilisera des tampons auxiliaires comme un z-buffer, un pochoir, etc. Pour un contexte utilisant deux tampons d'affichages, un mode RGBA et un z-buffer on utilisera :

```
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);
```

– `glutInitWindowPosition(int x, int y)` et `glutInitWindowSize(int width, int size)` : permet de spécifier la position (coin en haut, à gauche) et la taille de la fenêtre (en pixels).

– `int glutCreateWindow(char *string)` : crée une fenêtre avec un contexte OpenGL. Retourne l'identificateur de la fenêtre. La fenêtre ne sera affichée que lors de l'appel à la fonction `glutMainLoop()`.

2.8.2 Contexte OpenGL

Une fois la fenêtre crée par la fonction `glutCreateWindow`, un contexte OpenGL lui est associé. On peut donc alors (et pas avant) fixer certains états de la machine OpenGL (comme, par exemple, activer la gestion des faces cachées).

2.8.3 Fonctions associée aux évènements

Différents types d'évènements sont gérés par la GLUT. L'évènement incontournable étant la demande d'affichage. À chaque évènement on associe une fonction qui sera appelée en réponse de celui-ci. Ainsi la fonction associée à l'évènement de demande d'affichage contiendra les commandes OpenGL permettant le rendu de la scène dans le tampon d'affichage.

On spécifie l'association évènement-fonction de traitement par des fonctions de la forme glutEventFunc. On trouve, entre autres :

– `glutDisplayFunc(void (*func)(void))` : permet de spécifier la fonction qui sera appelée à chaque demande d'affichage. En cas d'utilisation du tampon des profondeurs, la fonction de traitement doit se terminer par un appel à `glutSwapBuffers()` permettant d'intervertir les deux tampons d'affichages.

– `glutReshapeFunc(void (*func)(int w, int h))` : fonction appelée lorsque la fenêtre change de taille. `w` et `h` sont les nouvelles largeur et hauteur de la fenêtre.

- `glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))` : fonction appelée losqu'une touche a été frappée au clavier. `x` et `y` sont la position de la souris au moment de la pression de la touche.
- `glutMotionFunc(void (*func)(unsigned char key, int x, int y))` : fonction appelée losqu'un bouton de la souris a été pressé ou relaché.

2.8.4 L'attente

Une fois les différents paramètres réglés on demande à la GLUT d'entrer dans une boucle infinie à l'écoute des évènements grâce à la fonction :

```
glutMainLoop()
```

2.9 Un exemple

Nous donnons un exemple de programme OpenGL utilisant la GLUT (OpenGL Utility Toolkit) comme gestionnaire de fenêtre. Nous décrirons le fonctionnement de la GLUT ainsi que ses principales fonctions lors des TD et des TP.

```
/** Mon premier triangle **/

#include<GL/glut.h>

/** Initialisation de la machine OpenGL **/
void myInit(void) {
  /* Couleur de fond : noir */
  glClearColor(0.0,0.0,0.0,0.0);

  /* On définit la matrice de projection
     (ici l'identité) */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  /* On va travailler par la suite sur
     la matrice modèle-vue **/
  glMatrixMode(GL_MODELVIEW);
}

/** Affichage de la scène **/
void display(void) {

  /* On efface ce qu'il y avait avant */
  glClear(GL_COLOR_BUFFER_BIT);

  /* On place l'objet
     (on travaille sur la matrice modèle-vue)
     - on réduit le triangle de 2 dans toutes les directions
     - on tourne le triangle selon l'axe des z
```

```
     - enfin, on le déplace de .5 selon l'axe des y */
  glLoadIdentity();
  glTranslatef(0,.5,0);
  glRotatef(45,0,0,1);
  glScalef(.5, .5, .5);

  /* On dessine le triangle */
  glBegin(GL_TRIANGLES);
  glColor3f(0,1,0);
  glNormal3f(0,0,-1);
  glVertex3f(-1,-1,0);
  glVertex3f(1,0,0);
  glVertex3f(-1,1,0);
  glEnd();

  /* On force l'affichage */
  glFlush();
}

int main(int argc, char *argv[])
{
  /* Initialisation de la GLUT */
  glutInit(&argc, argv);
  /* On demande un contexte OpenGL en mode RGBA */
  glutInitDisplayMode(GLUT_RGBA);
  /* Initialisation de la fenêtre */
  glutInitWindowSize(400,400);
  glutInitWindowPosition(0,0);
  glutCreateWindow("Triangle");
  /* Initialisation de la machine OpenGL */
  myInit();
  /* On définit la fonction qui sera appelée lors
     d'une demande d'affichage */
  glutDisplayFunc(display);

  /* Boucle principale */
  glutMainLoop();

  return 0;
}
```

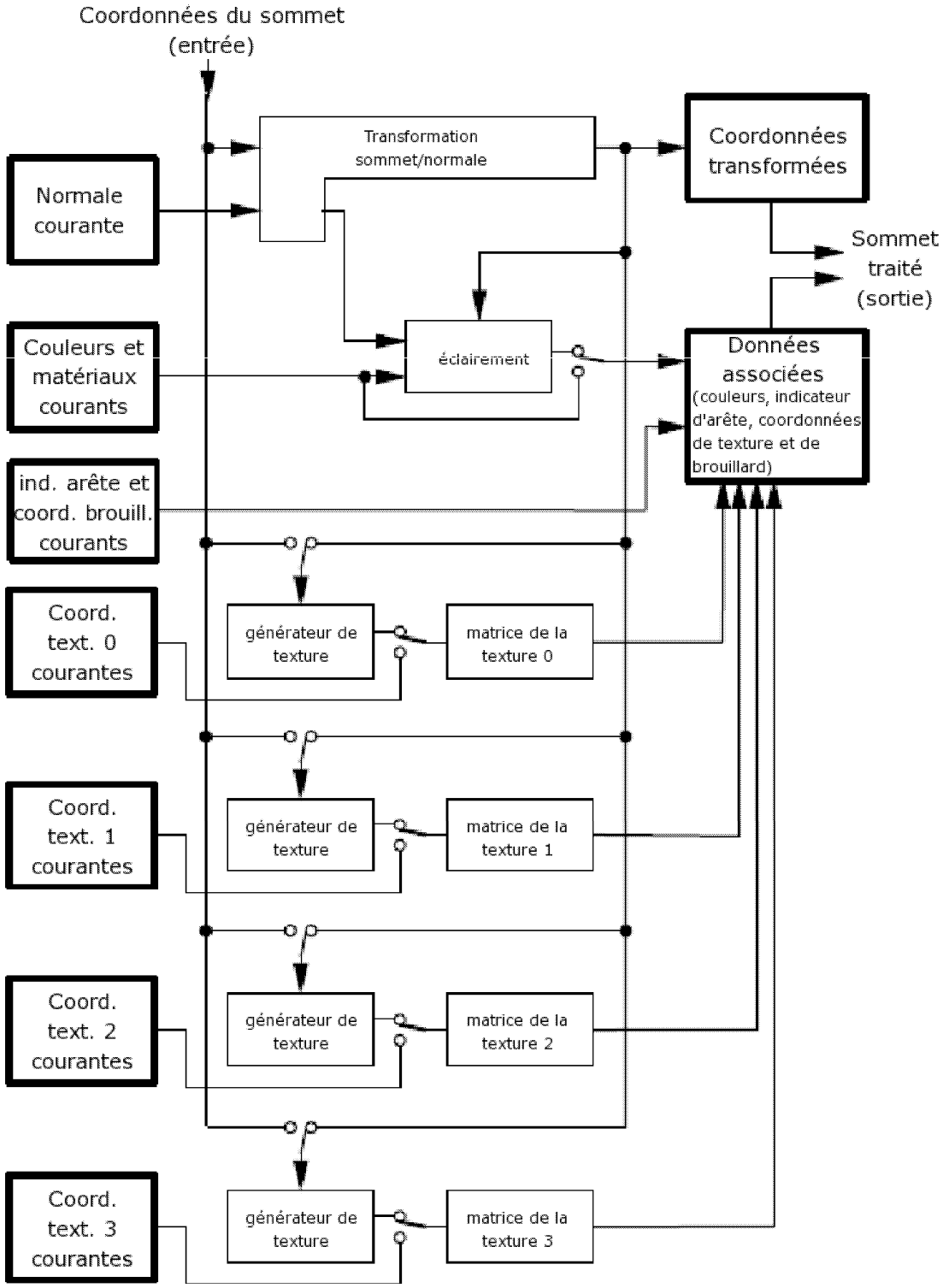

FIG. 2.1 – *Schéma de construction d'une primitive*

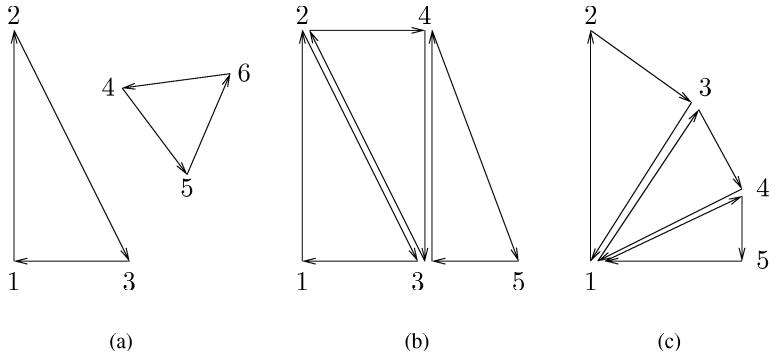

FIG. 2.2 – *(a)* `GL_TRIANGLES`, *(b)* `GL_TRIANGLE_STRIP`, *(c)* `GL_TRIANGLE_FAN`

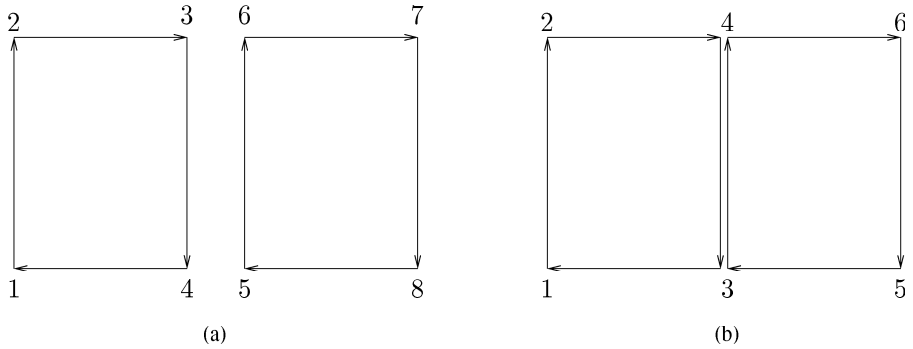

FIG. 2.3 – *(a)* `GL_QUADS`, *(b)* `GL_QUAD_STRIP`